Calculating Voxel-Polyhedron Intersections for Meshing Images

Stephen A. Langer*

Information Technology Laboratory

National Institute of Standards and Technology

Gaithersburg, MD, 20899

Andrew C.E. Reid

Material Measurement Laboratory

National Institute of Standards and Technology

Gaithersburg, MD, 20899

(Dated: June 12, 2019)

Abstract

Finite element meshes constructed from 3D images are useful in material science and medical applications when it is necessary to model the actual geometry of a sample, rather than an idealized approximation of it. Constructing the mesh involves computing the intersection of the mesh elements with the voxels (3D pixels) of the image. If done naively, this process is unstable, and small errors in the computed position of an intersection point can lead to large errors in the computed volume. We demonstrate the source of the instability and present a robust and efficient method of doing the computation, based on the r3d algorithm of Powell and Abel.

I. INTRODUCTION

When computationally modeling a complex object, it is often convenient to begin with an image of the object. For example, in materials science, a material may comprise many different grains with different shapes, compositions, and orientations, as shown in Fig. 1. Simulating the behavior of a 3D region provides insight into the behavior of the material as a whole. Basing the simulation on a micrograph of the material ensures that it accurately reproduces the geometry of (at least) one particular instance of structure.[1] Similarly, a medical image might provide the real geometry of a bone, and a simulation based on the image could compute the bone's strength and other properties.[2]

The Object-Oriented Finite element (OOF) software developed at the National Institute of Standards and Technology (NIST) uses finite element analysis to compute properties of complex materials, starting from experimental or simulated micrographs. When the project began, creating 3D micrographs was difficult, tedious, and uncommon, so OOF1[3] and OOF2[4] worked only in 2D, using 2D physical approximations. 3D micrographs are now common, and the third major revision of OOF, OOF3D[5], creates 3D meshes and uses 3D physics. This paper describes part of the mesh generation method in OOF3D. However, the method is applicable to any 3D technique for meshing images.

Mesh generation in OOF begins by first segmenting an image, assigning material properties to each pixel, and/or putting pixels into user-defined groups. (We will sometimes use the words "pixel" and "voxel" interchangeably, to avoid excess verbiage.) For the purposes of this paper, all that matters is that pixels can be categorized, and that pixels in one group are somehow different from pixels in another group. For a finite element mesh to be an accurate representation of the geometry of the image, one requirement is that each element be as *homogeneous* as possible – that is, it should overlie pixels of only one group, considering that pixels are rectangles and voxels are rectangular prisms, not points. A typical inhomogeneous element is shown in Fig. 2. Note that it is always possible to create a completely homogeneous mesh by creating a single element per pixel, but this mesh would be overrefined – it would resolve unphysical pixel corners and contain far more elements than are usually required for a sufficiently accurate computation.

A simple way of computing the homogeneity of an element E would be to count the number of pixel centers within the element for each pixel category. If there are $n_i(E)$ such

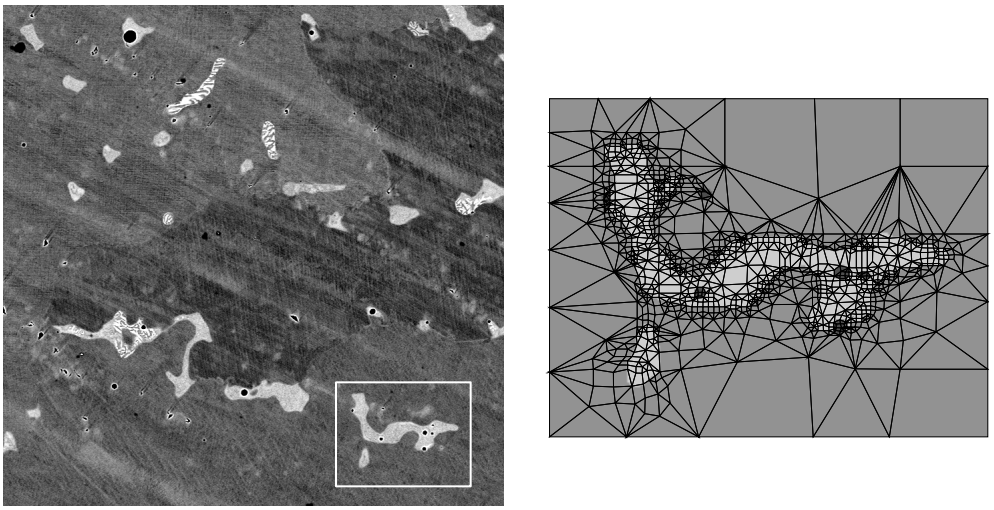

FIG. 1. A scanning electron microscope image (a) of a cast cobalt-chrome alloy with metal carbide inclusions, approximately 200 μm on a side. The light gray inclusions have a different composition than the darker background, which consists of two regions with the same composition but different crystalline orientations. The diagonal stripes and dark spots are artefacts of the sample preparation process. Image courtesy of Adam Creuziger at NIST. The region in the white rectangle has been meshed in (b) by OOF2.

pixels of category i, then the homogeneity could be defined as

$$h_{\text{poor}}(E) = \frac{\max_i n_i(E)}{\sum_i n_i(E)}. \tag{1}$$

This simple definition clearly fails when an element is small enough that very few (or zero) pixel centers lie within it. Furthermore, it produces a homogeneity that is a discontinuous and piece-wise constant function of the positions of the element's nodes, which is undesirable for many mesh modification tools. For example, a tool which moves nodes will find that many small moves don't change the homogeneity at all, providing no guidance on whether or not moving a node in a given direction will improve the mesh.

A more robust way of defining the homogeneity is

$$h(E) = \frac{\max_i V_i(E)}{V(E)} \tag{2}$$

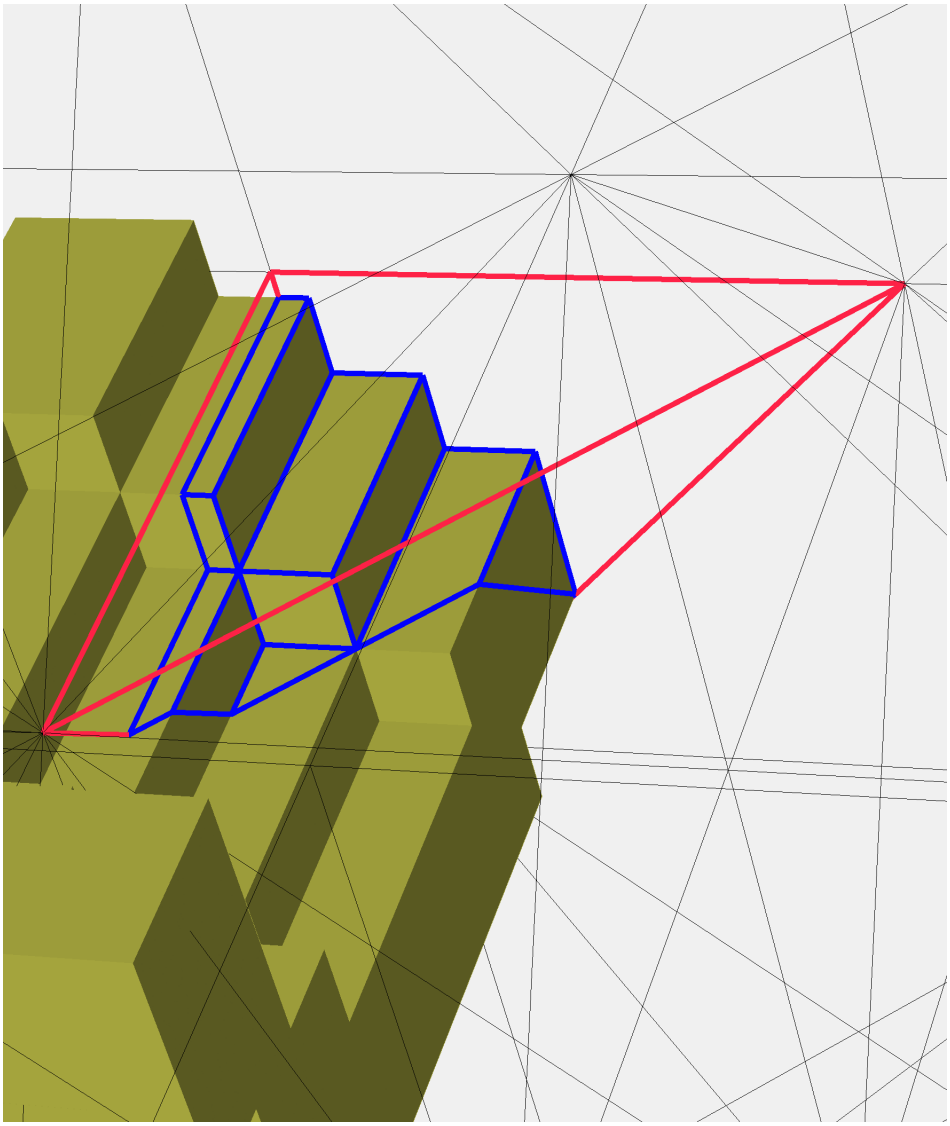

FIG. 2. A screen shot from OOF3D. The yellow voxels have the same material type, which is different from the material of the the other, invisible, voxels. The thin lines are the edges of the elements in a uniform tetrahedral mesh. The element highlighted in red is inhomogeneous because it contains both yellow and invisible voxels. The blue edges have been computed by the method described here and outline the intersection of the yellow voxels and the red element.

where $h(E)$ is the homogeneity of element E, and V_i is the volume (in 3D) or area (in 2D) of the part of the element that intersects pixels of category i. The total volume (or area) of the element is $V = \sum_i V_i$. The definition (2) of the homogeneity has the advantage that it is a continuous function of the positions of the element's nodes. Its main disadvantage is that it is far more complicated to compute than the simple definition (1).

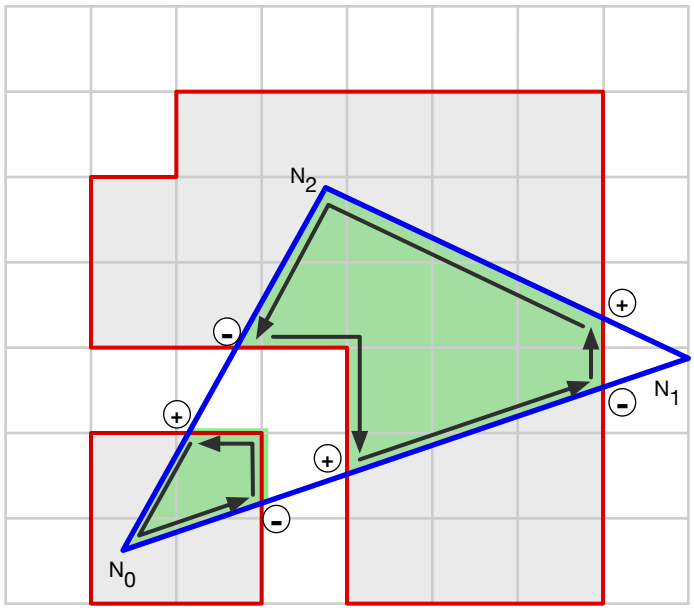

FIG. 3. Finding the area of intersection of an element and a set of pixels in 2D. Intersections between the pixel set boundary (red) and element boundary (blue) are marked as entrances (+) or exits (-). Entrances are connected to exits counterclockwise along the element boundary and elements are connected to exits counterclockwise along the pixel set boundary (black arrows).

In the remainder of this paper we first show how a naive way of computing Eq. 2 can run into trouble, and then introduce a method based on the r3d algorithm of Powell and Abel.[6] We show how the r3d method can fail in some cases, and how to fix it. Finally we discuss a method for efficiently computing the polyhedral graphs required by the algorithm.

II. WHY IS THIS HARD?

The simplest way of computing the volume of the intersection between a set of voxels and an element is to consider each voxel independently, and sum over them at the end. This is slow and inefficient, even in 2D, and we won't discuss it further. A more attractive algorithm is to extract the boundary of the set of voxels, and compute its intersection with the boundary of the element. In this way one can compute each facet of the intersection region, and therefore its volume. OOF1 and OOF2 use this method, although it is not without its subtleties.

For simplicity, we will demonstrate the method and one of its failure modes in 2D. Fig. 3

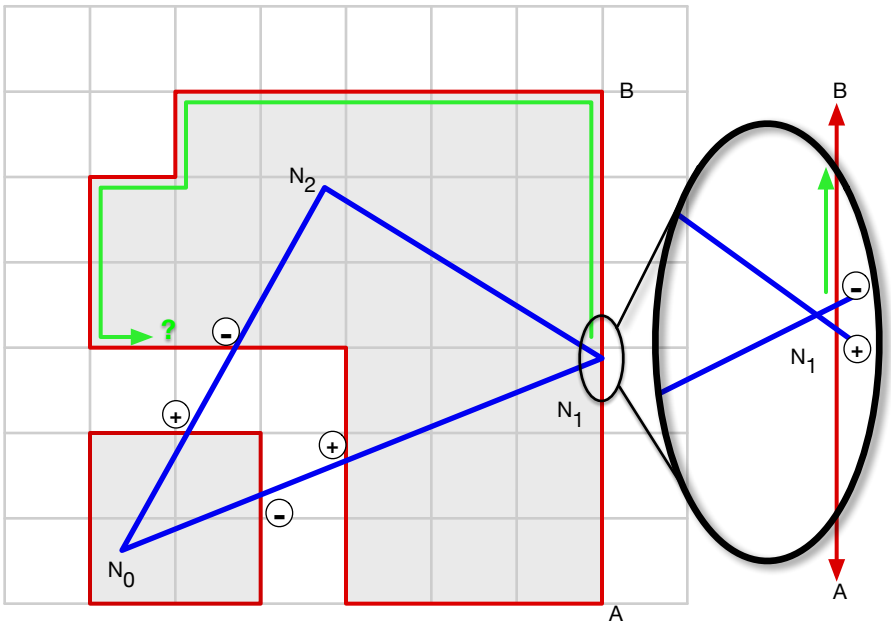

FIG. 4. Failing to find the area of intersection of an element and a set of pixels in 2D. The entrances and exits at node N_1 are in the wrong order, so the green segments leaving the exit are incorrect.

shows a triangular element intersecting a set of pixels (which need not be simply connected). We want to compute the area, $\mathcal{A}$, of the shaded intersection, $\mathcal{I}$ (which need not be simply connected). First, all of the pixels of a given type are aggregated, and their boundary is extracted, as shown in red. Then the intersection points between the segments of the pixel boundary and the segments of the element boundary are detected. If the element boundary (shown in blue) is traversed counterclockwise, each intersection point can be labelled as an entrance or an exit, depending on whether the element boundary enters or exits the pixel set at that point. Then the new boundary segments of the intersection can be constructed by following the element boundary from an entrance to the next exit, and following a pixel set boundary from an exit to the next entrance. To compute the area, the segments don't even need to be linked into loops. If the segment i goes from point $\mathbf{S}_i^0$ to $\mathbf{S}_i^1$, the area is

$$\mathcal{A} = \frac{1}{2}\sum_{i=0}^{N}(\mathbf{S}_i^0 - \mathbf{C}) \times (\mathbf{S}_i^1 - \mathbf{C}) \tag{3}$$

for some conveniently chosen center point $\mathbf{C}$. To avoid loss of precision in the sum, $\mathbf{C}$ should not be too far away from the segments.

Now consider what happens if the rightmost node of the element in Fig. 3 coincides with the pixel set boundary at location $\mathbf{N}_1$, as shown in Fig. 4. Note that this is not at all an

unlikely situation, since any algorithm that makes elements homogeneous will tend to put nodes on pixel boundaries. There are a number of things that can go wrong. Round-off error in the computations can lead to inconsistent information about whether point $\mathbf{N}_1$ is inside or outside the pixel set, since it may make it seem that segment $\mathbf{N}_0\mathbf{N}_1$ intersects segment $\mathbf{AB}$ but $\mathbf{N}_1\mathbf{N}_2$ doesn't (or vice versa). Or if the program is smart enough to insist that $\mathbf{N}_1$ must be either inside or outside, but not both, it can still misplace intersection points on segment $\mathbf{AB}$, as shown in the inset. It's possible that node $\mathbf{N}_1$ may be outside the element (by an infinitesimal amount) but that the computed entrance intersection of segment $\mathbf{N}_1\mathbf{N}_2$ with $\mathbf{AB}$ might *precede* the exit intersection of $\mathbf{N}_0\mathbf{N}_1$ on $\mathbf{AB}$. In this case, following the pixel boundary from the the exit point to the entrance won't connect the two points, but will follow the green line, adding incorrect edges to the perimeter of $\mathcal{I}$. Thus a microscopic error in the position of the points can lead to a macroscopic error in the area $\mathcal{A}$.

In 3D, the method and its failure modes are similar, except that both are more complex. We want to find the volume, $\mathcal{V}$, of the intersection, $\mathcal{I}$, of a set of voxels with a convex polyhedral element. First, the exterior of a set of voxels is divided into a set of facets. Each facet is a set of pixels in a plane. The intersection of that plane with the element is a polygon, and the intersection of that polygon with the pixels in the plane can be found using the 2D algorithm. These in-plane intersections form part of the exterior of $\mathcal{I}$. The remaining faces of $\mathcal{I}$ lie on the faces of the element, and can be computed by finding the edges of the in-plane intersections that lie on the element faces, and linking them, if necessary, along the edges of the faces. All of the difficulties of the 2D method apply, with additional complications. Is a point actually on a face? Actually on an edge? Which face is it on? What if a segment appears to intersect only one face of the element but both of its endpoints appear to be outside?

In 2D, these difficulties can be overcome by careful bookkeeping and using simple geometry to resolve ambiguities, following a "topology first" scheme. For example, in the example of Fig. 4, the fact that two exits (-) appear consecutively is a clue that something is wrong.[7] In 3D, detecting and correcting errors devolves into a forest of special cases and the resulting code is fragile, inefficient, and untrustworthy.

Related problems in computational geometry have been solved by different methods, such as arbitrary precision arithmetic.[8] This paper presents an alternative approach that avoids the issue of round-off error completely.

III. POWELL AND ABEL'S R3D ALGORITHM

An algorithm by Powell and Abel,[6] based on ideas from Sugihara,[9] provides a robust way of computing the intersection volume. Powell and Abel's r3d algorithm is a way of conservatively transferring data from one mesh to another so that integrals of the data over the new mesh are the same as integrals over the old mesh. Their original paper[6] described a method that computed integrals of polynomials over the intersections of two convex polyhedra (the polyhedra being elements from the old and new meshes), but a modified algorithm[10] works even if one of the polyhedra is not convex. Importantly, the algorithm is not subject to roundoff error. Unlike the naive method discussed in Section II, small errors in arithmetic lead only to small errors in the results. We can use the method to compute the volume of the intersections $\mathcal{I}$ — our element is a convex polyhedron, our voxel set is a (possibly) non-convex polyhedron, and the polynomial that we are integrating is 1.

r3d works by constructing a planar graph of the non-convex polyhedron and clipping it successively by the plane of each face of the convex polyhedron. The graph is isomorphic to the non-convex polyhedron — nodes of the graph correspond to vertices of the polyhedron, edges of the graph correspond to edges of the polyhedron, and they are connected identically. (We will use the word "vertex" to refer to corners of the polyhedron, and "node" to refer to the corresponding points in the graph.) If the polyhedron is sliced by a plane that divides it into two (possibly more, if nonconvex) regions, r3d constructs the graph of the new polyhedron on one side of the plane, using only information about which vertices are on which sides of the plane. Powell and Abel explain that by preserving topological consistency at all steps, and not relying on any arithmetic operations after deciding which vertices to clip, the method is robust in the presence of numerical errors.

To be specific, when intersecting polyhedron $\mathcal{P}$ with convex polyhedron $\mathcal{E}$, the process is[10]:

1. Construct a planar graph of $\mathcal{P}$, making sure that the edges at each node are in the same order as the edges of the polyhedron at the corresponding vertex, when viewed from outside the polyhedron. Figs. 5a and c show a polyhedron and its planar graph. A planar graph is one that can be drawn in a plane with no crossing edges. Each node must link to exactly three edges. (If a vertex of the polyhedron is on more than three edges, that vertex must be represented by more than one node in the graph,

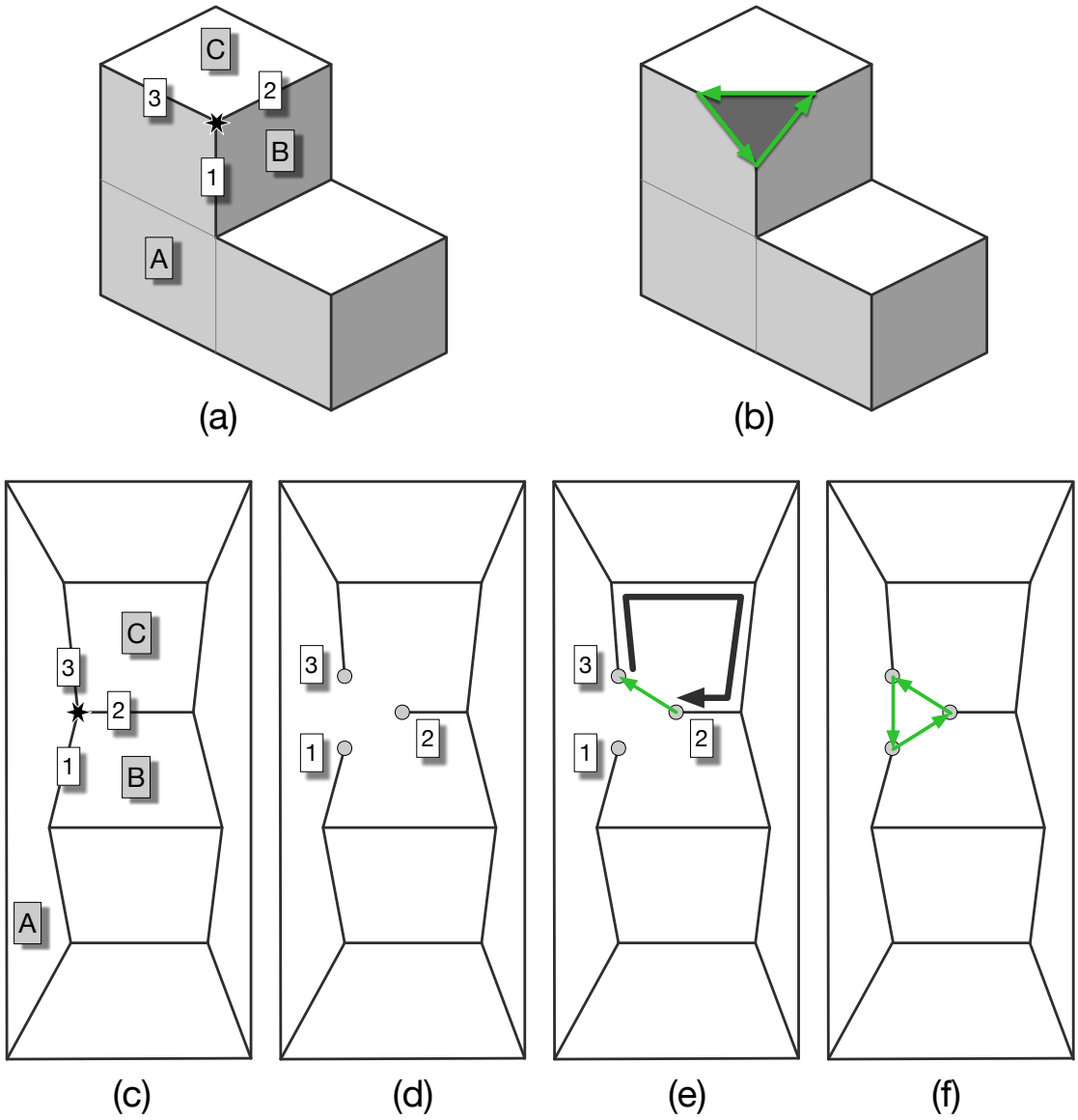

FIG. 5. Clipping a single corner from a polyhedron composed of three voxels. (a) The unclipped polyhedron, with three faces and edges labelled. The clipping plane is not shown, but passes between the starred vertex and the rest of the polyhedron. (c) The graph of the polyhedron, with corresponding faces and edges labelled. (d) The clipped node has been removed from the graph, and new dangling nodes (circles) inserted. (e) The endpoints of a missing edge are discovered by following graph edges from one dangling node to another, turning right at all intermediate nodes. (f) The completed graph, after all dangling nodes are completed. (b) The clipped polyhedron, reconstructed from the graph.

possibly connected by edges of zero length.) The edges of the graph divide the plane into regions that correspond to the faces of the polyhedron. Every convex polyhedron can be represented by a planar graph.

2. Choose one of the faces of $\mathcal{E}$. The clipping plane is the plane containing this face.

3. Remove the vertices of the graph that are outside the clipping plane, leaving dangling edges, as shown in Fig. 5d. Put a new node at the end of each dangling edge. The position of the corresponding vertex on the polyhedron can be found by interpolating along the original unclipped polyhedron edge. If rounding error puts an intersection point just past the end of the edge, it can be assumed to coincide with the endpoint. The graph itself contains no position information, so in the graph the position of the node is immaterial.

4. Choose one of the new nodes, $\mathbf{N}'$. In Fig. 5e, this is node 3.

5. From $\mathbf{N}'$, follow existing edges from node to node, always picking the right hand branch at pre-existing nodes (all of which have three edges) until another new vertex, $\mathbf{N}''$, is encountered. In Fig. 5e, this is vertex 2. It makes sense to refer to the "right hand branch" because the graph is planar.

6. Add a directed edge from $\mathbf{N}''$ to $\mathbf{N}'$, as shown in green in Fig. 5. The direction can be used later to determine the outward normal of the face, but is not strictly necessary.

7. Go back to step 4 and repeat until all nodes have three edges, as in Fig. 5f.

8. Go back to step 2 and repeat for each face of the element $\mathcal{E}$.

9. Find the facets of the clipped polyhedron by picking an edge of the graph and moving from edge to edge, turning left at each node, until the original edge is reached. From the edges, compute the area and normal vector of each facet, and from that the volume.

When the voxel set has concavities, it is possible that a clipping plane will intersect it in more than one region. Fig. 6 illustrates one such situation and the application of the algorithm to it. When both of the starred vertices in Fig. 6a are clipped, the intuitive solution is to replace each with a triangular facet. The algorithm instead generates two linked triangular facets, shown in Fig. 6b. However, the new vertices at points 3, 1, 4, and

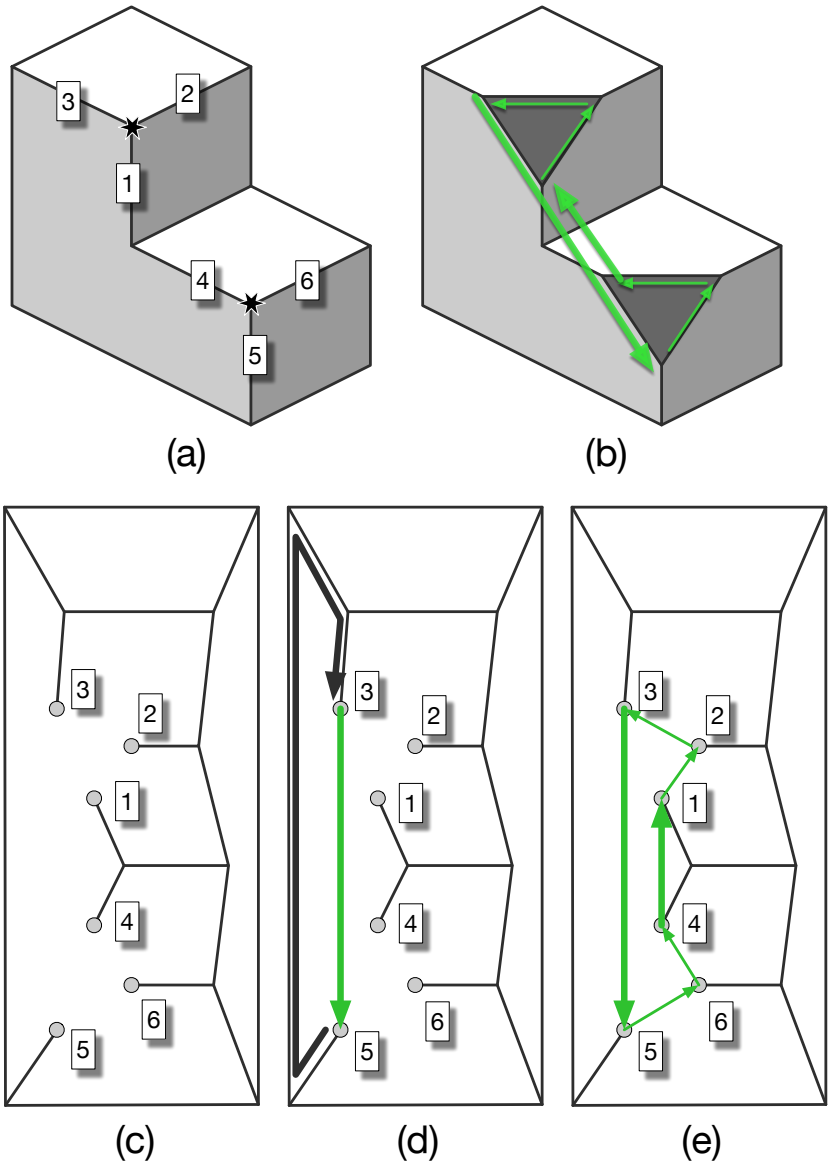

FIG. 6. Clipping two corners from the polyhedron of Fig. 5. (a) The unclipped polyhedron. The starred vertices are outside the clipping plane and will be removed. (c) The polyhedral graph with the clipped nodes removed. The unclipped graph is the same as Fig. 5c. (d) Following the graph from new node 5 to new node 3 along the heavy black line creates an edge (green arrow) that appears to span too much of the graph. (e) The completed graph. (b) The clipped polyhedron. The heavy arrows indicate collinear edges of the new facets. The edges cancel each other out in the region between the new triangular facets (dark gray).

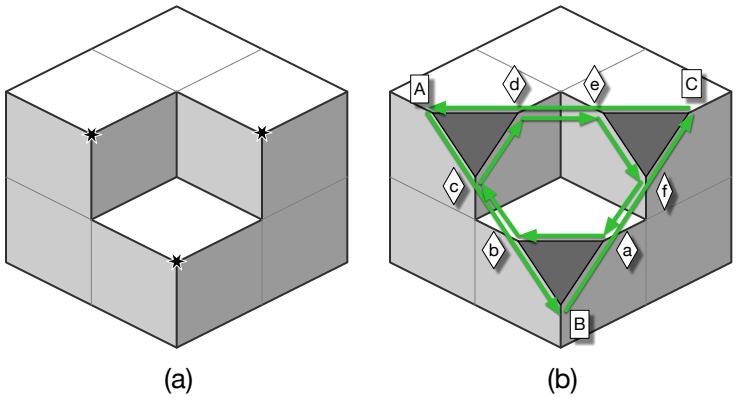

FIG. 7. Clipping another concave set of voxels. (a) A set of 7 voxels, arranged in a cube with one corner missing. The corners marked with stars are outside the clipping plane and will be removed. (b) The result of applying the r3d algorithm. There are two new faces denoted by the green arrows. The faces are coplanar and oppositely directed, so the inner hexagonal face (*abcdef*) cancels the interior of the outer triangular face (*ABC*).

5 are collinear (on the line formed by the intersection of the left face of the voxels and the clipping plane), so between points 1 and 4 the bridge between the triangles comprises two oppositely directed collinear segments. These make no net contribution to the area of the facet or volume $\mathcal{I}$ of the clipped polyhedron, so this somewhat convoluted geometry is harmless. (If the vertices were *not* collinear, the left face would necessarily be split into multiple facets, which would introduce new edges in the graph, and the algorithm would no longer connect vertices 3 and 5.) Similarly, in Fig. 7 a concave set of voxels is clipped so that three corners are removed. In this case, there are two new coplanar facets, a triangle and a hexagon, with oppositely directed normals. The triangular facet is indicated by the heavy green arrows in the figure. The hexagonal facet, indicated by the thin arrows, cancels the area of the interior of the triangular facet, so that net result is the three dark triangular faces. Using directed edges when reconnecting the clipped graph makes it clear that the hexagonal and triangular faces have opposing normals. Again, the result is correct for the purpose of computing the volume of the clipped polyhedron.

To maintain topological consistency, r3d relies on the graph of the polyhedron being 3-vertex-connected, that is, not separable into two pieces by removing any two nodes. Polyhedra created from voxel sets can easily fail this test, but the algorithm still works with a slight modification. Consider the polyhedron shown in Fig. 8, built from a $2 \times 2 \times 4$ brick of

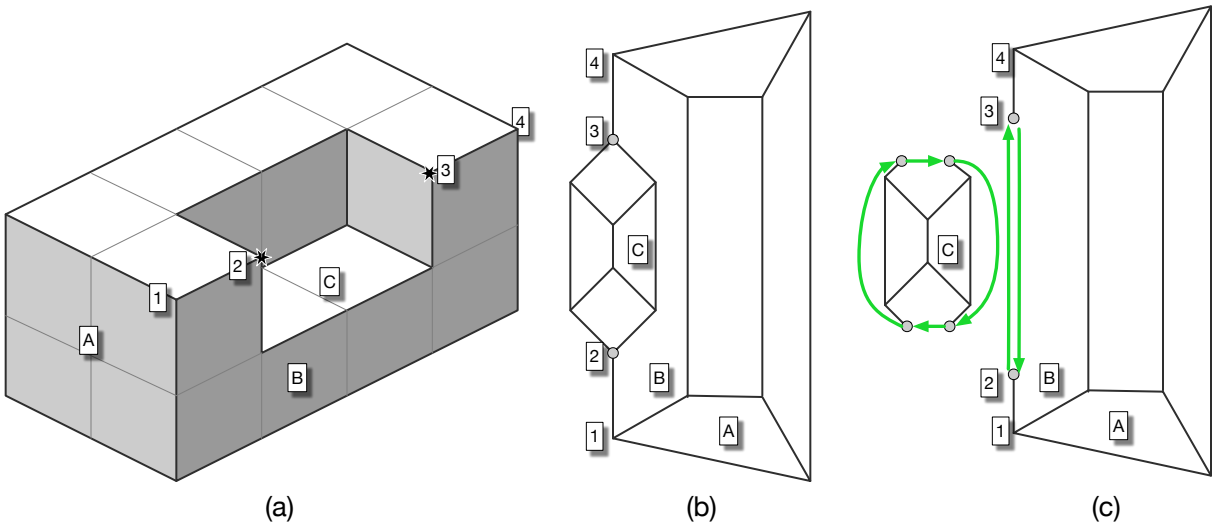

FIG. 8. Applying r3d blindly to an insufficiently connected polyhedron. (a) The voxels forming the polyhedron. The vertices marked with stars are outside the clipping plane. (b) The graph of the polyhedron, with some nodes and faces labelled for comparison with (a). (c) The result of applying r3d to (b).

voxels, with 2 voxels removed from one side. The graph of this polyhedron, Fig. 8b, is not 3-vertex-connected, because it falls into two pieces if nodes 2 and 3 are removed. Now clip the polyhedron with a plane that removes only vertices 2 and 3. This can only occur due to numerical error, because a plane that clips vertices 2 and 3 must necessarily also clip either vertex 1 or vertex 4 or both, but this is exactly the sort of numerical error that is expected and must be handled. Applying r3d to Fig. 8b results in the disconnected pair of graphs shown in Fig. 8c, with the new edges shown in green. The left hand part, containing face C, is an inverted (negative volume) pentagonal prism. It is difficult to interpret the right hand graph. Do the two oppositely directed edges cancel each other out? Does that leave dangling nodes at their ends?

The fact that this situation must be the result of numerical error indicates how it should be resolved. Vertices 1, 2, 3, and 4 must actually be collinear. (Any modification to the polyhedron that makes them non-collinear without adding extra vertices between them also restores 3-vertex-connectivity.) Infinitesimally perturbing the polyhedron by shaving a strip off of the corner, as shown in Fig. 9, will not change its volume. The perturbed polyhedron has two new faces, labelled 'a' and 'b', and its graph has 3-vertex-connectivity. Using r3d to clip the *four* starred vertices produces the disjoint pair of graphs in Fig. 8c. The left

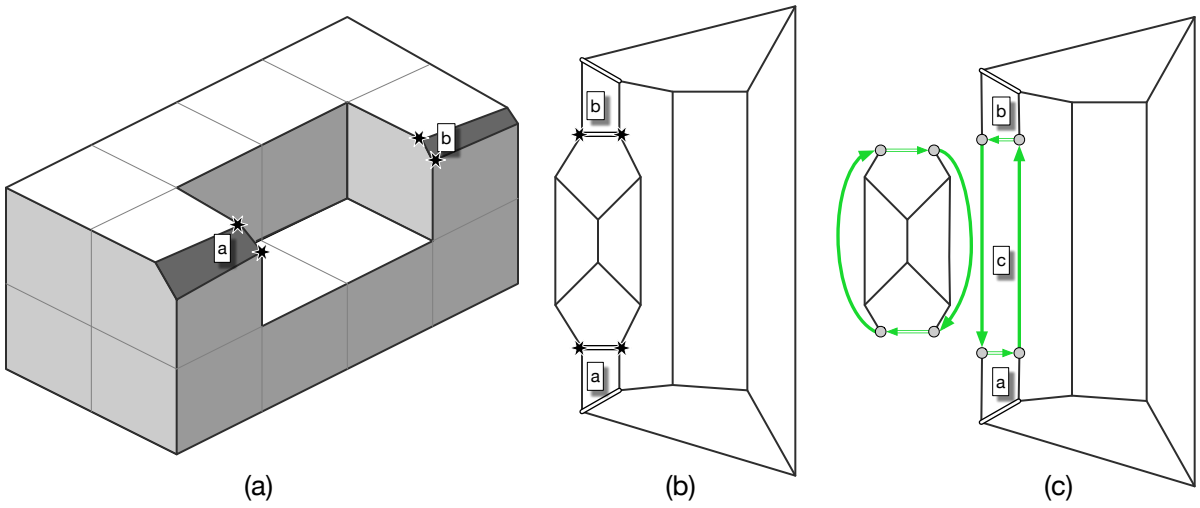

FIG. 9. A perturbed version of Fig. 8. (a) The perturbed polyhedron, with two new infinitesimal faces. Stars mark the vertices to be clipped. (b) The graph of the polyhedron. The doubled edges are infinitesimal. (c) The result of applying r3d to (b).

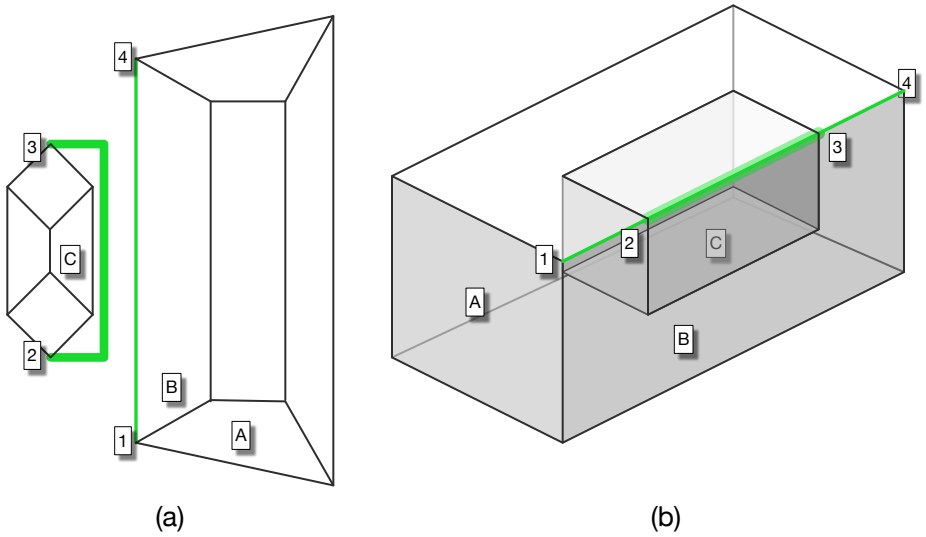

FIG. 10. Resolution of the problems in Fig. 8. Vertices 1 and 4 are joined directly (thin green line) as are vertices 2 and 3 (thick green line). The inner polyhedron is subtracted from the outer one.

hand graph is again an inverted pentagonal prism, one face of which has two infinitesimal edges. The right hand graph is a pentagonal prism with an infinitesimally thin facet that is broken into three sub facets labelled a, b, and c, all positively oriented. Collapsing the infinitesimal edges and faces leads to the graph and polyhedron in Fig. 10. (The arrows have been dropped from the new edges, because they no longer belong to new faces whose

orientation must be determined.) The eight vertices on the thin facet in Fig. 9c are all collinear so the facets collapse into a single line from vertex 1 to vertex 4. In other words, the correct stratagem is not to connect nodes 2 and 3 in Fig. 8c with a double edge, but to connect nodes 1 and 4 with a single edge. The final graph is two separate quadrilateral prisms, one of which is inverted. The polyhedron is formed by the subtraction of the smaller polyhedron from the larger, and its net volume is the same as the initial set of voxels in Fig. 8, as it should be.

Computationally, it is difficult and expensive to verify 3-vertex-connectivity and to know when this procedure will need to be imposed. However, it is fast and easy to check for doubled edges in the graph, and to repair them by replacing them and the single edges on either side (*i.e,* 1234 in Fig 8c) with one single edge (14 in Fig 10a).

IV. CONSTRUCTING GRAPHS OF VOXEL SET BOUNDARIES

Before r3d can be applied to a segmented image, the graph of each voxel set must be found. This can be a time consuming procedure, but it can be done just once, and doesn't need to be repeated unless the voxel categories change. Here we discuss how this is done in OOF3D, using C++.

A. Algorithm Components

It's convenient to first describe the classes that we use to represent the voxel set boundary and its graphs.

A `VoxelSetBdy` describes a voxel set boundary. It creates a graph from an image and computes intersections.

A `VSBGraph` is the graph of the boundary edges of a voxel set. It is contained in a `VoxelSetBdy`.

A `VSBNode` is a node in a `VSBGraph`. Each `VSBNode` knows its neighboring nodes, and its position in space.

A `ProtoVSBNode` is a point in the image at a corner where voxels meet, and encodes the geometry of the voxel set at the corner.

The algorithm constructs a `VSBGraph` of the voxel set boundary. The first step of the algo-

rithm is to identify relevant voxel corners, which are associated with `ProtoVSBNode` objects. The `ProtoVSBNode` object sub-types reflect the local geometry, which in turn determines how many (zero or more) `VSBNode`s are required for each `ProtoVSBNode`. Finally, using the geometry encoded in the `ProtoVSBNode`s, the appropriate edges are inserted between the corresponding `VSBNode`s, the `ProtoVSBNode`s are discarded, and the graph is now complete.

B. Creating Proto Nodes

The first step in building a graph is constructing an array of `ProtoVSBNode` (Proto Voxel Set Boundary Node) pointers, one for every voxel corner in the image. If the image is $k \times l \times m$ voxels, the array size is $(k+1) \times (l+1) \times (m+1)$ because the `ProtoVSBNode`s live at the corners of the voxels, not the centers. Each `ProtoVSBNode` contains information about the eight voxels that meet at that point in the image. A voxel is *occupied* if it's in the current category. (On the image edges, the non-existant voxels outside the image are considered to be unoccupied.) Assigning a 1 to an occupied voxel and 0 to an unoccupied voxel and arranging the bits in an arbitrary but consistent order reduces the local configuration of eight voxels to a one byte signature. (See Fig. 11.) There are different `ProtoVSBNode` subclasses for different signatures. Although there are 256 signatures, only 17 `ProtoVSBNode` subclasses are needed, because voxel configurations that can be rotated into one another are represented by a single class. For example, one `ProtoVSBNode` subclass covers all of the eight configurations that have only one occupied voxel. Another subclass covers the 24 orientations of the configuration of three occupied voxels in Fig. 12. A `ProtoVSBNode` of the correct subclass is created at each voxel corner using a signature-based lookup table, storing the rotation required to bring the actual voxel configuration into alignment with the subclass's reference orientation. A `ProtoVSBNode` is not allocated for the 38 configurations of voxels that don't correspond to a corner in the aggregated voxel set.

C. Creating Graph Nodes

The second step is to create a new empty `VSBGraph` object, and for each `ProtoVSBNode` to create one or more `VSBNode`s in it. The `VSBGraph` is mostly just a container for `VSBNode`s. A `VSBNode` stores its position in the image and pointers to its three neighboring `VSBNode`s in the

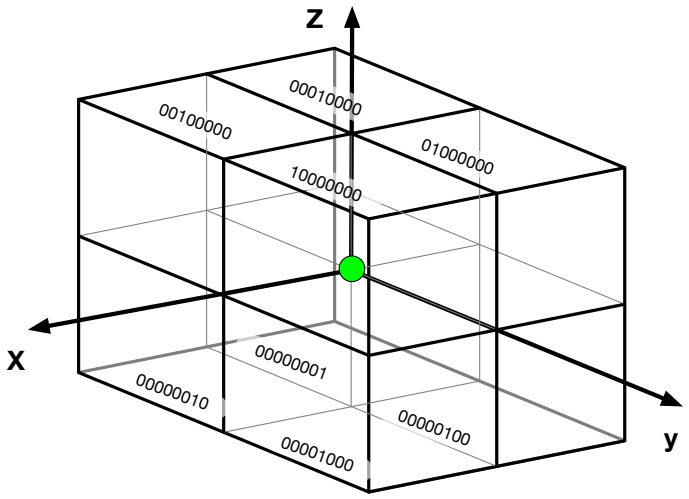

FIG. 11. The geometry of a `ProtoVSBNode`. The green circle is the position of the node. The binary digits are the labels of the eight neighboring voxels. A voxel is *occupied* if it's in the category whose graph is being computed. The *signature* of the `ProtoVSBNode` is the bitwise-or of the labels of the occupied voxels.

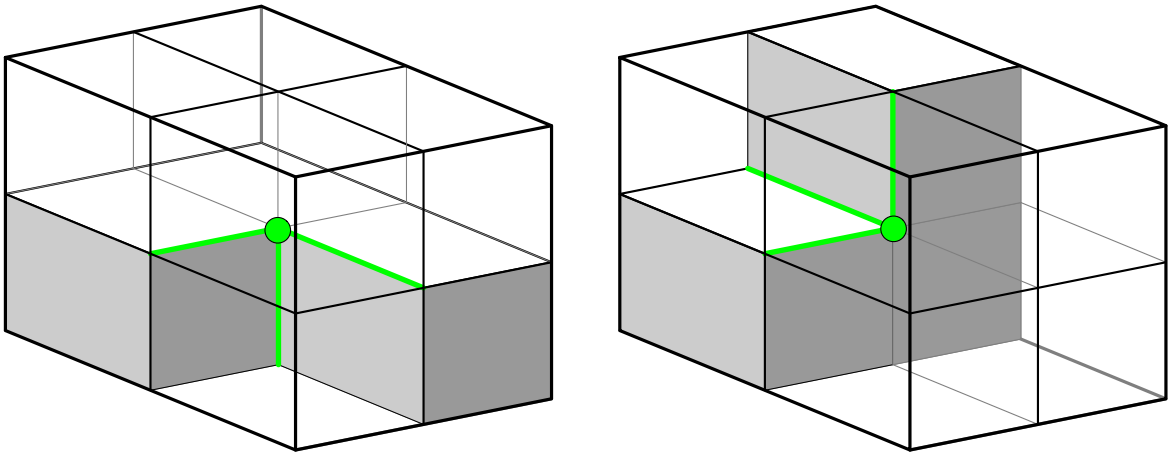

FIG. 12. Two of the 24 voxel configurations in one of the `ProtoVSBNode` subclasses. The green circle is the position of the `ProtoVSBNode`, and the occupied voxels are shaded. The green lines show the directions in which this `ProtoVSBNode` needs to look for neighboring `ProtoVSBNode`s when connecting the graph.

graph (thereby mixing information from the real-space and graph-space representations of the polyhedron). In an unclipped graph, a `VSBNode`'s position is at integer voxel coordinates, but node positions in a clipped graph are not constrained and must be stored as floating point numbers. The list of pointers to neighbors is ordered so that neighbor $(i + 1) \bmod 3$

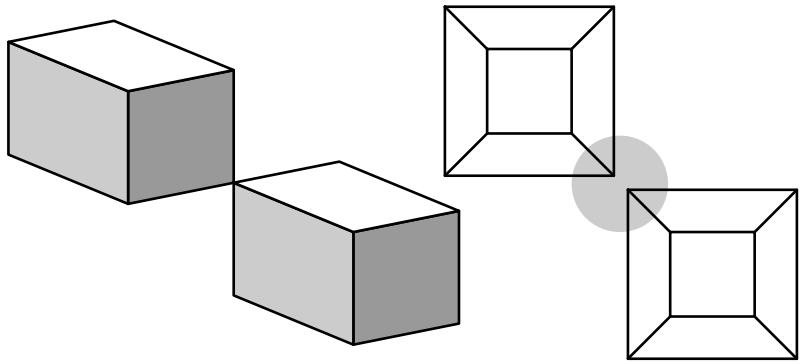

FIG. 13. A polyhedron formed from two voxels that touch at a corner, and its graph, which consists of two independent sections. The central vertex of the polyhedron is represented by two nodes (drawn inside the shaded circle) in the graph.

is clockwise from neighbor i, which facilitates graph traversal in the r3d algorithm.

A single `ProtoVSBNode` must create more than one `VSBNode` when there are more than three edges that meet at a point. A simple example is the case of two voxels (or any two rectangular blocks of voxels) that touch at a corner, as shown in Fig. 13. The graph of this polyhedron has two independent parts, although one node in each part shares its position with a node in the other. The `ProtoVSBNode` at that point creates two `VSBNode`s and links to six edges.

The colocated graph nodes in Fig.13 are not connected directly to one another, and should not be connected because they don't share faces of the polyhedron. Some `ProtoVSBNode`s, however, require the addition of multiple `VSBNode`s connected to one another by edges of length zero. One example is shown in Fig. 14.

D. Connecting Graph Nodes

The third step in constructing the graph is to add edges connecting the `VSBNode`s. The `ProtoVSBNode`s at each point know in which directions they connect, so they can search in those directions for their nearest neighbors. The two `ProtoVSBNode`s at either end of a edge then collaborate to decide which of their `VSBNode`s should be connected to one another, and, importantly, how the edges must be inserted into the `VSBNode`s so that they are in the correct order for r3d.

The `ProtoVSBNode`s do all of the work in determining how `VSBNode`s should be connected,

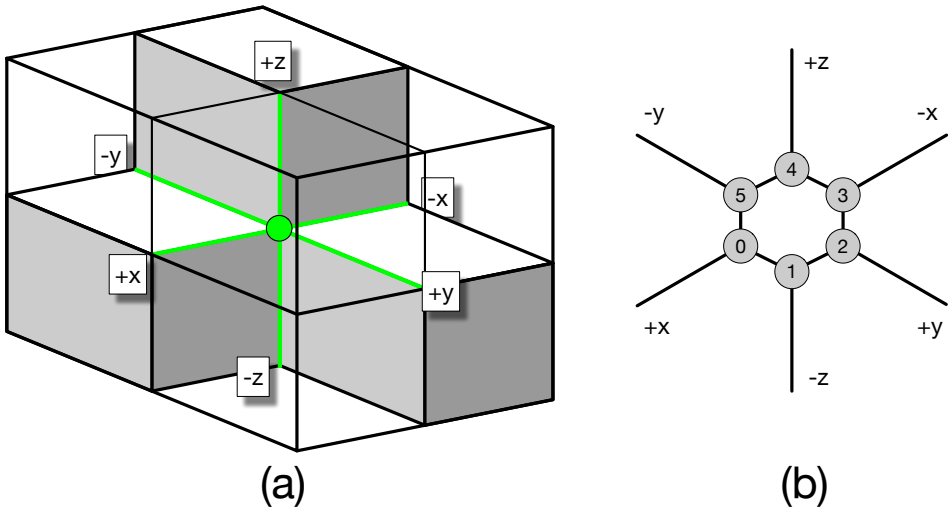

FIG. 14. (a) A `ProtoVSBNode` for a configuration of four voxels (one is hidden) in its reference orientation. The green edges are the ones that connect to neighboring nodes. (b) A section of the graph created by the `ProtoVSBNode`. Six `VSBNode`s are required. The edges between them have zero length. The numbers indicate the indexing used internally to ensure that the correct edges are connected to the nodes.

because they have complete information about the local voxel structure. Each `ProtoVSBNode` subclass knows which of its `VSBNode`s connect to the graph edges in each direction in its reference orientation. For example, Fig. 14b shows the configuration of nodes and edges for the `ProtoVSBNode` shown in its reference orientation in Fig. 14a. The `VSBNode` numbered 0 connects to the `VSBNode`s numbered 1 and 5 and to a `VSBNode` created by the next `ProtoVSBNode` in the `+x` direction.

The actual connection process falls into one of three categories, described in the following subsections.

1. Connecting a Single Pair of Nodes

Connecting two `VSBNode`s in the same `ProtoVSBNode` is trivial, because the `ProtoVSBNode` contains all of the information that it needs. The simplest non-trival case is when two `ProtoVSBNode`s must connect their `VSBNode`s along a direction that requires a single edge (ie, a situation like that in Figs. 12 or 14 but not Figs. 15 or 16). For two `ProtoVSBNode` objects `A` and `B` to figure out which of their `VSBNode`s connect along which edges, `A` calls

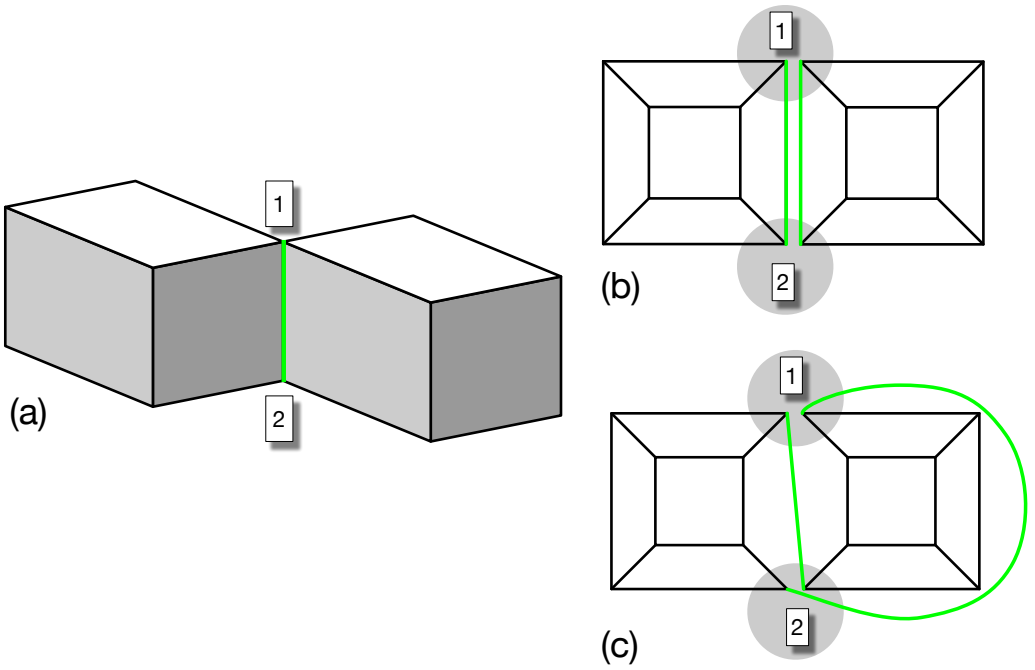

FIG. 15. A situation requiring both coincident nodes and coincident edges in the graph. The two blocks of the polyhedron (a) could be single voxels or blocks of voxels. They share the green edge. The graph (b) is the same as the graph of two cubes, although the two nodes at point 1 coincide, as do the nodes at point 2. Both green edges correspond to the green edge in (a). When constructing the graph, it is important that the nodes are connected correctly. Misconnecting nodes along the doubled edge as in (c) results in a graph that is not 3-vertex-connected.

`B.connect(A)`. `B` finds the spatial direction from itself to `A`, rotates the direction into its (`B`'s) reference frame, and thereby knows which connection is being made. For example, if `B` is the `ProtoVSBNode` in Fig. 14, and `A` is in direction `+y` in `B`'s reference orientation, then it's connecting to `B`'s node number 2. `B.connect` knows which of its `VSBNodes`, `bnode`, needs to be connected, but it doesn't know which of `A`'s to connect to. To find out, it calls `A.connectBack(B, bnode)`, which finds the appropriate `VSBNode` (`anode`) in `A`, inserts `bnode` in the correct slot, and returns `anode`, which `B.connect` can now insert in the correct neighbor slot in `bnode`. (Simple!) `ProtoVSBNode.connect` and `ProtoVSBNode.connectBack` are virtual functions, defined independently in each `ProtoVSBNode` subclass, enabling the connection process to depend on the local geometries of both `ProtoVSBNodes`.

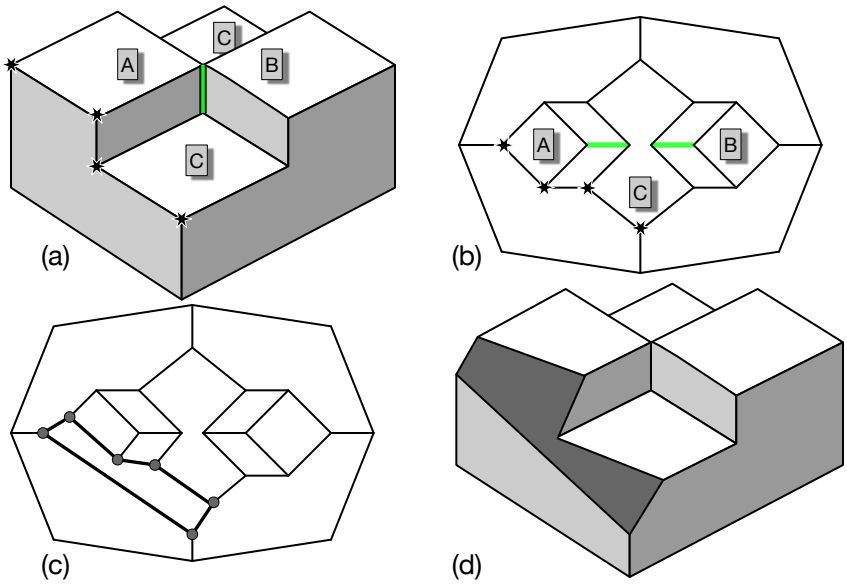

FIG. 16. (a) A polyhedron that yields a doubled graph edge, as in Fig. 15, but in a non-trivial environment. The green edge in the polyhedron, where two voxels touch edge-wise, will become two edges in the graph. The two facets labeled C are coplanar. (b) The graph of the polyhedron, with the green edges from (a) highlighted. (c) The clipped graph, after the vertices marked with stars in (a) and (b) are removed. The heavy lines and circles mark the new vertices and edges. (d) The clipped polyhedron.

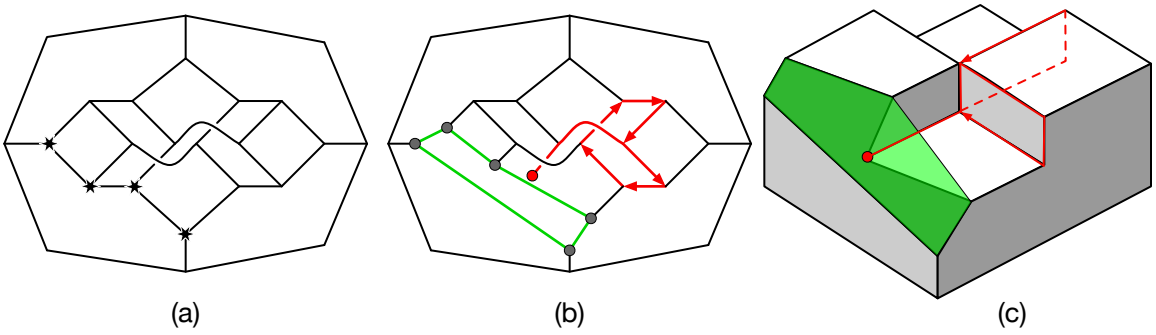

FIG. 17. (a) The nodes on the doubled edge in the polyhedron in Fig. 16 are connected incorrectly. The graph is nonplanar. (b) The result of using r3d to remove the starred nodes from (a). The circles represent new nodes. Green lines show one plausible but incorrect facet. The red lines show r3d's path through the graph when starting at the red node. The path returns to its starting node, indicating that the algorithm has failed. (c) The polyhedron represented by (b), drawn on top of the correctly clipped polyhedron from Fig. 16. The incorrect path is marked in red.

2. Connecting a Double Pair of Nodes

Some varieties of `ProtoVSBNode` need to create two graph edges in the same spatial direction. Fig. 15 shows a simple example. There there are 5 edges at each of the vertices marked 1 and 2, which are both represented by two graph nodes. The vertices share the same location in physical space, but the nodes are distinct in the graph topology. If the wires are crossed, so to speak, the resulting graph may not have the required connectivity and may produce incorrect results when clipped. Fig. 15c is not 3-vertex-connected, but as it happens this configuration doesn't cause a problem for r3d. Fig 16, on the other hand, shows a case in which misconnection of the edges produces a graph for which r3d results in nonsense, as seen in Fig. 17. We should not expect r3d to work on Fig. 17a, because the graph is not planar, but this illustrates the importance of ensuring that the connection procedure guarantees planarity.

Note that the two graph edges that leave the two `VSBNode`s in each of the `ProtoVSBNode`s always lie along the edges of two voxels that share an edge, as in Figs. 15 and 16. Call these the *key* voxels. Thinking of the key voxels as having an infinitesimal gap between them, each of the two graph edges follows the edge of its own voxel, and connects to the graph node associated with the same key voxel at both `ProtoVSBNode`s. The key voxels must have the same relative spatial positions in both `ProtoVSBNode`s (in real space, not necessarily in the reference orientation of the `ProtoVSBNode`), or else there would be another `ProtoVSBNode` intervening between `A` and `B`. Therefore, the correct way to connect the `VSBNode`s is to find the corresponding key voxels in each `ProtoVSBNode` and connect the `VSBNode`s that belong to the edges of those voxels. However, `A` and `B` may be in different `ProtoVSBNode` subclasses, with different orientations, and do not know the correspondence between their sets of key voxels.

If the key voxels at `A` are i and j, let p_i^A and p_j^A be their real space coordinates, and let N_i^A and N_j^A be the corresponding `VSBNode`s. p_i^A and p_j^A are vectors with integer coordinates, and we define an arbitrary but self-consistent greater-than operator for such vectors. (We are guaranteed that $p_i^A \neq p_j^A$.) Because the key voxels in `B` must have the same relative position as the key voxels in `A`, $p_i^A > p_j^A \iff p_k^B > p_l^B$ if k and l are the key voxels in `B` that correspond with i and j in `A`. This tells us how to connect the nodes. In the program, `A.connect(B)` finds `A`'s key voxels and passes the ordered pair to a virtual function, `B.connectDoubleBack(A,`

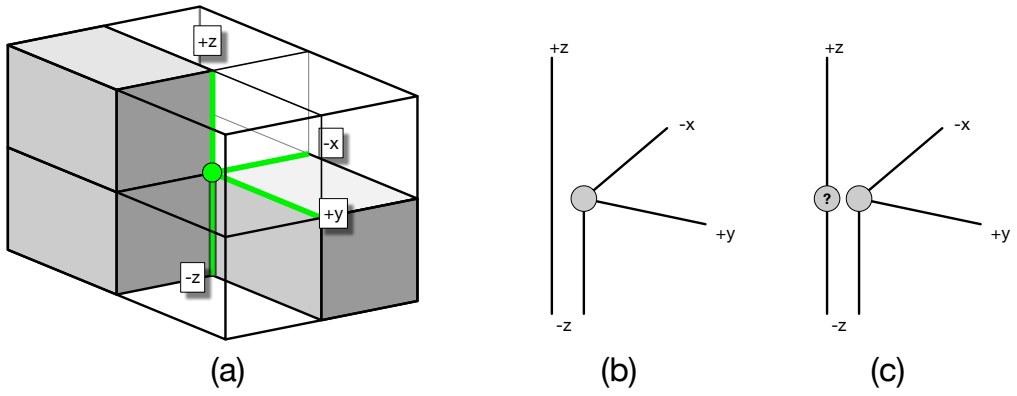

FIG. 18. A special case. The `ProtoVSBNode` (a) needs to create a single `VSBNode` with three edges (b) and also allow neighboring `VSBNode`s in the $\pm z$ directions to connect directly to one another. Creating a temporary `VSBNode` (c) with only two edges allows the connection process to involve only nearest neighbor `ProtoVSBNode`s.

`N0, N1)`, where `N0` is a pointer to the `VSBNode` whose key voxel has the larger position and `N1` is a pointer to the other. `B.connectDoubleBack` figures out which of its key voxels has the larger position, sets the neighbor pointers in its `VSBNode`s, and returns them as an ordered pair. `A.connect(B)` can then set the neighbor pointers in its own nodes correctly.

3. A Special Case

There is one more subtlety to the node connection procedure. The `ProtoVSBNode` configuration shown in Fig. 18a contains two stacked voxels with a third sharing an edge with one of the other two. The `ProtoVSBNode` needs to create one `VSBNode` connecting along all three edges of the third voxel. Another graph edge passes through the `ProtoVSBNode` along the edge of the stacked voxels, but doesn't connect to a `VSBNode` there. This causes a problem, because that extra edge means that this `ProtoVSBNode` sits between two other `ProtoVSBNode`s that need to connect to one another, but `ProtoVSBNode`s always connect to their nearest neighbors. The expedient solution is to allow it to create a `VSBNode` with only two edges, as shown in Fig. 18c. Then after all connections are made, those nodes and their edges can be removed and replaced by a single edge.

E. Cleaning Up

Finally, after all connections are made, all of the `ProtoVSBNode`s can be deleted.

V. ACCURACY CHECKS

To check the accuracy of the algorithm, we used OOF3D to create artificial microstructures containing two categories of voxels and superimposed finite element meshes on the microstructure. We calculated the volumes of the elements in two ways: V_g is the geometric volume, using just the positions of the nodes, while V_r uses r3d to find the intersection of the element with each voxel category, and sums over categories. For each element we found the relative error, $E = |V_r - V_g| / V_g$, and computed its average, $\bar{E}$, standard deviation, δE, and maximum over the entire mesh, $E_{\max}$. In addition, we computed the total volume, V_t^i of each voxel category i by summing the r3d results in each element and also by simply counting the number of voxels N_i in each category in the entire microstructure, multiplied by the volume of a voxel. The relative discrepancy between these two results, $E_i = |V_t^i - N_i V_{\text{vox}}| / (N_i V_{\text{vox}})$ gives another error estimate for each category.

All tests were performed on microstructures that were cubes of side 1 (in arbitrary units). The cubes were divided into $M \times M \times M$ voxels. A uniform finite element mesh was created by dividing the microstructure into $N_x \times N_y \times N_z$ cubes, and splitting each cube into 5 tetrahedral elements. Usually, we took $N_x = N_y = N_z \equiv N$.

As a baseline check, we used only one category of voxel, making all intersection calculations trivial. With $M = 50$ and N ranging from 1 to 50, $\bar{E}$ increased roughly linearly from 6.3×10^{-16} to 1.5×10^{-14}, with $\delta E \approx \bar{E}$ at each N. The largest value of $E_{\max}$ was 1.6×10^{-13}. The error in the total volume of the category, E_0, was larger and noisier as a function of N, but still always below 7×10^{-12}. (Machine epsilon was 2×10^{-16}.)

Nontrivial tests were done with two different microstructure geometries, both using two categories of voxels. In the first geometry, voxels were randomly assigned to one of two categories with probability p. In the second geometry, all voxels within a set of randomly placed and possibly overlapping spheres were in one category, and voxels outside the spheres were in another. The number of spheres, their mean radius, and the width of the radius distribution were adjustable. All randomized configurations were repeated at least 20 times.

Using M of 5, 50, and 100, and N of 4, 10, 11, no errors greater than 10^{-14} were observed in any runs of either geometry. The error magnitude increased when the elements' aspect ratio was allowed to vary, but even setting $N_x = 10000$, $N_y = N_Z = 1$ only raised the average error $\bar{E}$ to 1.3×10^{-12} and $E_{\max}$ to 1.2×10^{-11} (in a run with $M = 100$ and 60 spheres of mean radius 0.15).

The largest errors observed in any run occurred when the mesh was modified by moving nodes at random (while ensuring that elements were still well-formed). In one case, an element with a y-dimension that was 10^{-3} times its x and z dimensions had an error E of 3×10^{-10} (using a microstructure of 100 spheres with mean radius 0.1, $M = 100$, and $N = 4$). No larger errors were observed.

Given these results, it is reasonable to assume that errors in the calculation of element/voxel intersections will be negligible compared to other sources of error, such as finite element discretization or numerical error in the solution of PDEs.

VI. OPTIMIZATIONS

The elements of a mesh representing an image may be large on the scale of the voxels in the image, but are generally going to be small on the scale of the image itself. This means that if a graph is constructed from the whole image, a lot of time in the graph traversal process will be spent eliminating voxels that are far from the current element. The process can be made much more efficient if the image is first sliced up into bins that are a bit larger than the average size of an element, and separate graphs are created for each bin. Then only graphs whose bins intersect with the bounding box of the element need to be considered. A disadvantage of this method is that the graphs need to be recomputed occasionally if the mesh is refined and the average element size changes greatly.

If the image is very large, it may be inconvenient to create and store the full array of `ProtoVSBNodes`. It may be more efficient to create the `ProtoVSBNodes` on demand, and to delete them when they're known not to be needed. `ProtoVSBNodes` only need to look for neighbors in the positive x, y, and z directions. If the loop over nodes is done in the positive direction, all nodes with x positions less than the current value of x can safely be deleted. (This optimization has not yet been implemented in OOF3D.)

VII. OBTAINING THE CODE

The method described here is used in OOF3D but could be useful in other contexts in which convex polygons intersect image data. The implementation used in OOF3D is independent of other aspects of OOF3D so that it can be easily incorporated in other projects. It may be downloaded from `http://www.ctcms.nist.gov/oof/vsb`. OOF3D may be downloaded from `http://www.ctcms.nist.gov/oof/oof3d`. The voxel set boundary code is in OOF3D's `SRC/common/VSB` subdirectory. Both downloads are free.

VIII. ACKNOWLEDGMENTS

The authors would like to thank Devon Powell for helpful conversations and for sharing his software.

---

* stephen.langer@nist.gov

[1] R. Garcia, W. Carter, and S. Langer, Journal of the American Ceramics Society **88**, 750 (2005).

[2] S. Badilatti, G. Kuhn, S. Ferguson, and R. Müller, Journal of Orthopaedic Translation **3**, 185 (2015).

[3] S. Langer, E. Fuller, and W. Carter, Computing in Science and Engineering **3**, 15 (2001).

[4] A. Reid, S. Langer, R. Lua, V. Coffman, S.-I. Haan, and R. Garcia, Computational Materials Science **43**, 989 (2008). http://www.ctcms.nist.gov/oof/oof2.

[5] V. Coffman, A. Reid, S. Langer, and G. Dogan, Mathematics and Computers in Simulation **82**, 2951 (2012). http://www.ctcms.nist.gov/oof/oof3d.

[6] D. Powell and T. Abel, Journal of Computational Physics **297**, 340 (2015).

[7] Actually, even in 2D OOF2 uses a method similar to that described in this paper because it is simpler and requires no consideration of special cases. Instead of examining the entire element, it trims the pixel set one element edge at a time.

[8] J. Shewchuk, Discrete and Computational Geometry **18**, 305 (1997).

[9] K. Sugihara, Computational Graphics Forum **13**, 45 (1994).

[10] D. Powell, "r3d: Software for fast, robust geometric operations in 3d and 2d," (2015). https://github.com/devonmpowell/r3d/blob/master/la-ur-15-26964.pdf.